On Predicated Execution

Joseph C. H. Park, Mike Schlansker
Software and Systems Laboratory
HPL-91-58
May, 1991

control dependence, If-conversion, Modulo Scheduling, predicated execution, program dependence graph, software pipelining

Predicated execution as required in Modulo Scheduling technique for parallelizing innermost loops with conditional statements is examined. We obtain an improved semantics for predicate operations together with an optimal and efficient algorithm for their use.

1 Introduction

In compiling techniques, specifically *software pipelining*, for instruction-level parallelism, as in the loop scheduling technique (called Modulo Scheduling) of Cydrome due to Rau and others[4,9,10], Predicated Execution (simply, PE) consisting of If-conversion and certain architectural support (to be described) play an essential role. Given a control flow graph (simply, graph) representing a fragment of code, say, the body of an *innermost natural* loop, PE is used to remove all branching operations of the body thereby collapsing all blocks into a single block of straight line code. Succinctly, PE thus allows loops with conditional statements to be overlapped. Compared to other approaches the particular architectural support utilized distinguishes PE from a pure software approach such as Lam's Hierarchical Reduction in software pipelining[5].

An efficient method of If-conversion has been discovered by Ferrante et al.[3] in connection with Program Dependence Graph (PDG) proposed there. PDG has been studied as an intermediate representation yielding many benefits with respect to transformations that occur in parallelizing compilers. Here we derive for use in PE a variant of their technique pertaining to control dependence. Informally the meaning of a Control Predicate (simply, predicate) is such that blocks associated with the same predicate can be executed concurrently subject only to data dependence as soon as the predicate is defined to be *true*. Or, in a more picturesque language, nodes far apart in control flow are brought together in terms of control dependence.

In PE, If-conversion and control predicates are not merely compiling artifacts but their effects appear explicitly in parallelized machine code taking the form of Predicated Operations using Predicate Registers. For this purpose we have reexamined both the If-conversion technique of PDG as well as the existing method of the Cydrome compiler. A new result is obtained consisting of two parts, improved semantics of PE and improved algorithms for their use. Our method together with the new semantics cures a deficiency in the existing method of producing incorrect code for certain flow graphs. In addition, the new approach is superior with respect to the requisite number of predicate operations and their placement as well as to the algorithmic complexity itself.

The problem being studied can be phrased as follows. Basic blocks (simply blocks or nodes) of a graph are to be associated with predicates and operations that define predicates in use are to be added appropriately. We want to minimize both the number of predicates in use and the of defining operations necessary. That is, there are two parts to our problem:

- How to assign predicates to blocks.
- How to place defining operations for predicates in use.

We are thus led to capture solutions in terms of two functions, R and K, such that R prescribes assignment naming for each block the associated predicate and K prescribes placement directing for each predicate in use where and how it must be

defined. They are inherently different problems. Assignment is purely graph-theoretic in character, whereas placement depends, in addition, on (1) semantics of predicate operations and (2) style of execution as elaborated later.

We start by introducing relevant notations and concepts including the semantics of PE. We then give algorithms for determining R and K. This is followed by discussions.

2 Basic Notions

We assume readers are familiar with PDG work[3] and certain basic concepts, postdominator relation, control dependence, and the like. For the sake of breivty we do not repeat them here. As usual a control flow graph is a directed graph augmented with special nodes, Start and Stop. Every node is reachable from Start and can reach Stop.

A node (or a basic block) x is a sequence of straight line code where flow enters only from the top and leaves at the bottom in two different manner:

if (t_x) goto y else goto z, or
goto y

In the conditional case node x has two successors. The edge $x \rightarrow y$ is labeled true and $x \rightarrow z$, false. The associated branching condition t_x determines (during execution) which edge is taken. For the unconditional edge no label is required. Formally, an edges is thus either a triple or a pair:

(x, y, label), or
(x, y)

where label is true or false.

For technical reasons in computing control dependences as discussed subsequently we sometimes pretend that there are edges,

(Start, Head, true), and
(Start, Stop, false)

where the (unique) true successor of Start is named Head.

2.1 Semantics of Predicate Operations

We assume the architecture provides a set of predicate registers, each one bit in length, and two defining operations:

$p_y = \text{stuff}(t_x)\ \&\ p_x$, and (1)
$p_y = \text{stuffbar}(t_x)\ \&\ p_x$ (2)

where p_x is a source predicate register, p_y is a target predicate register, and t_x represents the result of evaluating the branching condition of block x.

The semantics of stuff operation is using self-evident notation (in an axiomatic style):

$$\begin{aligned} &\text{if } (p_x.\text{old})\ p_y.\text{new} = t_x.\text{old} \\ &\text{else } p_y.\text{new} = p_y.\text{old} \\ &t_x.\text{new} = t_x.\text{old} \\ &p_x.\text{new} = p_x.\text{old} \end{aligned} \tag{3}$$

Similarly for stuffbar:

$$\begin{aligned} &\text{if } (p_x.\text{old})\ p_y.\text{new} = \neg\, t_x.\text{old} \\ &\text{else } p_y.\text{new} = p_y.\text{old} \\ &t_x.\text{new} = t_x.\text{old} \\ &p_x.\text{new} = p_x.\text{old} \end{aligned} \tag{4}$$

Clearly these are merely predicated copy operations (albeit, from data to predicate register file and one involving complement) and their definitions are consistent with the notion of predicated operation in general described next.

The purpose of a predicate is to enable/disable (or nullify) an operation to which it is attached. Consider an operation predicated with a predicate p:

$$t_n = op(t_i,...t_j)\ \&\ p \tag{5}$$

The semantics is:

$$\begin{aligned} &\text{if } (p.\text{old})\ t_n.\text{new} = op(t_i.\text{old},...,t_j.\text{old}) \\ &\text{else } t_n.\text{new} = t_n.\text{old} \end{aligned} \tag{6}$$

leaving out trivial statements specifying no change. In particular, the meaning of a predicated operation is the same as No Operation, if the associated predicate is false.[1] Unpredicated operation is to be understood as operation predicated with constant true:

$$t_n = op(t_i,...t_j)\ \&\ \text{true}$$

2.2 Style of Predicated Execution

We turn now to the style of execution, or the underlying computing model specifying how predicated execution is to be used as reflected in the parallelized code. It will turn out that the placement problem (function K) depends on this choice of style. To facilitate our discussion, consider the graph shown in Fig. 1. In the figure assignments of predicates to blocks are shown using a short hand:

$$B_x(t_x)\ \&\ p_y$$

---

1. Their effect on schedule is, of course, not that of No Operation. In Modulo Scheduling as applied in overlapping iterations of a loop, operations of the loop body are scheduled taking all predicates to be true. Contrast this with Trace Scheduling, in which, whenever there is a choice one edge is selected to be true. In effect at compile time assertions about predicates are made, all true in one style and one each in the other. Performance in either approach is thus highly dependent on the likelihood of these assertions during execution.

denoting that (1) block B_x is associated with branching condition t_x and (2) every operations of B_x (including predicate defining operations, if any) are predicated with predicate p_y. For example, $B_3(t_3)$ & p_5, states that predicate register p_5 is assigned to block B_3 with branching condition t_3.

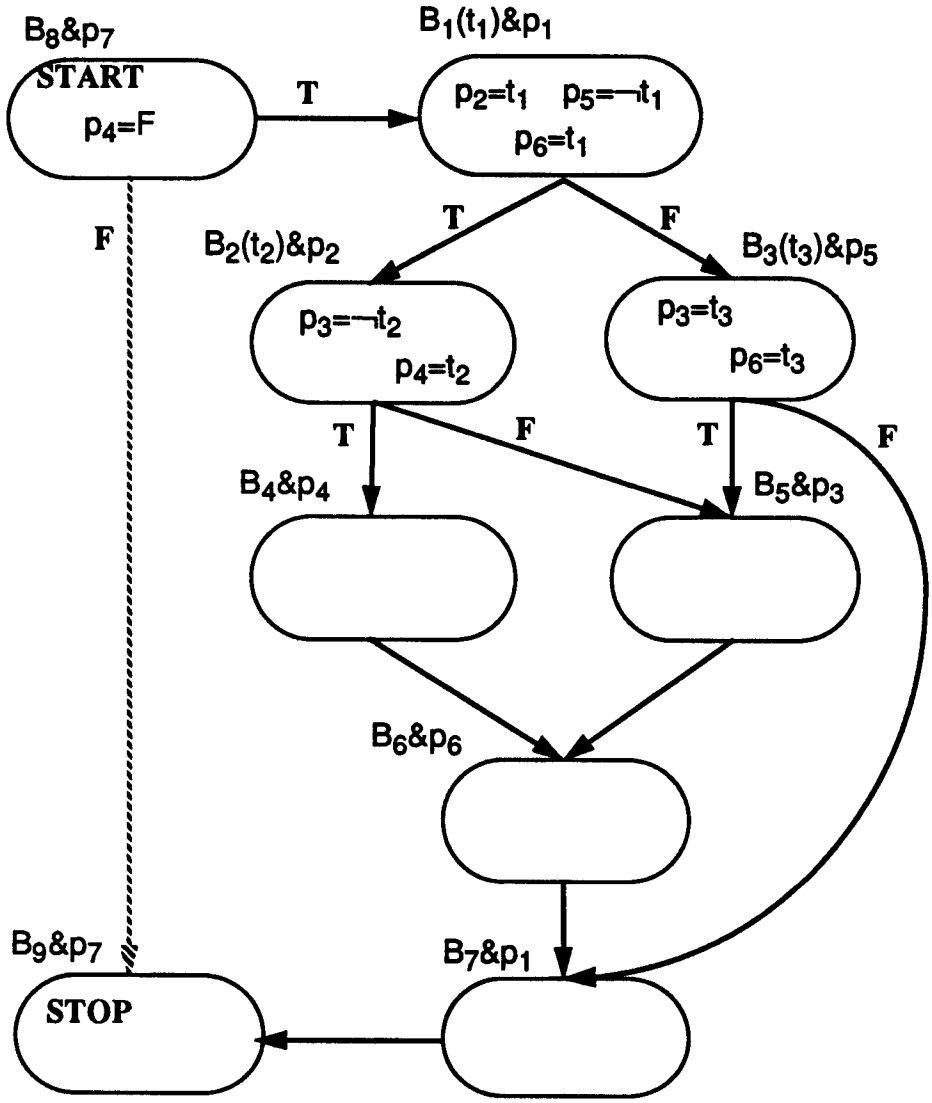


Figure 1. A control flow graph with predicate code. Predicates p_7 and p_1 are constant true.

Placement of defining operations are also shown in the figure abbreviated as

$$p_y = t_x$$

for stuff operation (1), and similarly $\neg t_x$ for the stuffbar operation (2). Note that the source predicate of a defining operation is implicitly that of the block where the operation is placed. Blocks show only predicate operations in the figure. Other operations of nodes are irrelevant in our discussion.

Effects of flow edges are entirely captured by predicates and their defining operations. Having eliminated control flow in this manner, we next want to choose blocks and merge them into a single block for the purpose of scheduling. In the style of interest, viz., Modulo Scheduling as applied in overlapping iterations of an *innermost natural* loop, all blocks (of a loop body) are chosen. Suppose in our example blocks are merged in the order,

$$B_1B_3B_2B_4B_5B_6B_7 \qquad (7)$$

The result represents a single block of straight line code. We shall refer to this as a *reduced* node. Remember that the body of an innermost natural loop is, by definition, acyclic and as usual, an acyclic graph is linearizes by topological sorting on control flow edges. In general, this embedding of partial order into a linear one is not unique. Other linear orders are possible and clearly it is desirable that the placement (K) be invariant under this embedding. As discussed later this feature is present in our approach, whereas it is precluded in the old approach due to a different semantics of the defining operations used.

With respect to such reduced code (7) the properties of PE we must have are:

- (P1) An operation is predicated with p_x iff its block is associated with predicate p_x.
- (P2) Block x associated with predicate p_x is enabled iff $at(x) \wedge p_x$.
- (P3) All predicates must be defined before their use.

The notation at(x), like the control predicate of Lamport[6], denotes that, when true, control (PC) is at the beginning of block x. We also use after(x) in the same vein with the understanding that after(x) = at(y), where y immediately follows x with respect to the topological order. The word "enabled" is used above with the meaning of "not nullified."

Properties listed appear obvious. Their implications, however, are not so trivial. Take a path from Start to Stop in our graph. If the underlying execution model allows branching as in the ordinary case, then clearly we need not impose any property on the nodes not in this path. On the other hand, in our style of execution, only one graph-theoretic path, and hence, one execution path exists, viz.,

$$\text{Start} \rightarrow (\text{reduced code}) \rightarrow \text{Stop}$$

involving all blocks of the original graph. That is, every execution involves *all* blocks. Hence, due to (P3), we must have all predicates defined in every execution including those that must be *false*. This is a stronger requirement on K than in any other style involving nodes fewer than all. In particular, K that would be suitable for control

flow model allowing explicit branching operations, does not satisfy (P3) and must be modified as discussed later.

To pursue the implications in detail, we simulate a (sequential) execution of the reduced code and consider its behavior as a sequence of states reflecting how predicates change. Semantics of predicates are fully captured by considering all such behaviors. As an example, take the reduced code (7) and consider its behavior for a case, t_1=false and t_3=true. We start from B_1 deliberately ignoring the reset operation, p_4=false, present in Start. The result is shown in Table 1.

Table 1. A Behavior of Reduced Code

p_X	after(B_X)	p_2	p_3	p_4	p_5	p_6	Remark
T	B_1	F	X	X	T	F	t_1 = F
p_5	B_3	F	T	X	T	T	t_3 = T
p_2	B_2						
p_4	B_4						p_4 = X at(B_4)!
p_3	B_5						
p_6	B_6						
T	B_7						

The table headings are, from left to right, predicate, block, and state (immediately after the execution of each block) showing only of predicates in use. Blocks are executed in the order of the reduced code (7). Entry "X" under predicates indicates "unknown". Missing entries under predicates denote "no change." This occurs either because the predicated associated with a block at hand is false or because the block does not contain defining operations affecting predicates under examination.

In the behavior shown we eventually have p_4 undefined at B_4. The code will or will not produce a correct result depending on the unknown value of p_4. This clearly shows that the reset operation, p_4 = false, in Start is necessary.

Given a graph the number of (*simple*) paths from Start to Stop is finite and so is the number of different (generalized) topological orders. Hence one can in principle verify a result by examining a finite number of such behaviors.[1] But this is superfluous, since our algorithm is proven correct.

1. 40 in our example

3 Our Formalism

In order to uncover the essential aspects of our problem quickly, we take as given and start from the control dependence function CD of PDG[3]. Let N be the set of nodes and C a set of control dependencies. We have a function

$$CD : N \rightarrow 2^C$$

such that for any node $x \in N$ CD(x) is the set of its control dependencies. In our notation each control dependence $c \in C$ is denoted as $\pm y$ with $+y$ (or simply, y) standing for the true edge leaving block y and -y for the false edge. Thus

$$CD(x) \equiv \{\pm y \in C: x \text{ is control dependent on } \pm y\} \quad (8)$$

The semantics of CD(x) is such that block x is enabled iff there exits $y \in CD(x)$ and the true (false) edge leaving y is an execution edge. The branching condition t_y of block y decides during execution whether or not true (false) edge is an execution edge.

In PE we capture entirely the effect of conditional branching by (1) appropriately placing in y the predicate defining operations

$$p = t_y, \text{ or } p = \neg t_y \quad (9)$$

(shown in abbreviated form) and (2) assigning predicates appropriately to blocks. Having done this all edges in the flow graph can be removed leading to the reduced code consisting essentially of one block, which is executed as a straight line code with the properties (P1) through (P3) stated earlier.

Our problem, when formally stated, is to compute two functions, R and K:

$$R : N \rightarrow P$$
$$K : P \rightarrow \text{range(CD)}$$

such that R solves the assignment problem and K the definition problem. For each node x in N a unique predicate p=R(x) is assigned. Predicate defining operations are inserted into nodes according to K. For example, when K(p)={+y, -z}, we place $p=t_y$ in node y (to be enabled/disabled with its own predicate p_y) and $p=\neg t_z$ in node z. Having done this for given R and K, we can remove all edges in the graph leading to the reduced code, which is unique upto topological sorting.

Aside from computing CD in the manner of PDG, the parts, new due to PE, of our algorithm, Algorithm RK, are (1) decomposition of CD into functions R and K and (2) augmenting K so that (P3) is satisfied. Rigorous arguments for its correctness are given elsewhere[8] and consists of two parts: One part that depends only on graph properties is straightforward. The remaining part that deals, in addition, with definition and use of predicates in PE is somewhat arduous[1] (specially in comparison with the simplicity of Algorithm RK). It involves formally capturing the essential (semantic) aspects of both control dependence and predicated execution (PE). This must be the case, since one (PE) entirely replaces the other (control dependence). The

1. This is no surprise: form is easy to deal with, meaning is not.

notion of control dependence is entirely absent in our reduced code consisting essentially of one node. We address in turn the decomposition problem, purely graph-theoretic, and the underlying meaning allowing the way we use R and K in PE. To facilitate discussion we will cite results from [8] without reproducing proof.

3.1 Decomposition of CD into R and K

The problem of obtaining R and K from CD is viewed best as a problem of partitioning N under a certain equivalence relation. We say that two nodes x and y in N are equivalent to each other if they have the same control dependence,

$$x \approx y \text{ iff } CD(x)=CD(y) \tag{10}$$

It is obvious that the relation $\approx$ defined thus is an equivalence relation on N. Let N be partitioned into equivalence classes P with respect to this relation. Let R be the function that maps each node x in N to the class p in P to which it belongs. This is denoted p = R(x). Let K be the function that maps each class p to CD(x), denoted K(p)=CD(x), where x is a representative of class p. It is clear that R and K are well-defined and unique upto renaming of classes in P.

By virtue of being a partition we have the desired properties:

- Every node x belongs to one and only one class p=R(x).
- Nodes with identical set K(p) of control dependencies belong to the same class p

We now summarize certain useful properties of R and K.

Lemma 1

K is isomorphism.

This lemma shows that K merely names each distinct member of range(CD). Since K is an isomorphism, its inverse exits and we can take R to be

$$R = K^{-1} \bullet CD \tag{11}$$

That is, for each node x R gives the name of the class the node x belongs. We now have the following result, which the decomposition step of Algorithm RK relies on.

Lemma 2

p = R(x) iff K(p) = CD(x)

The nodes, Start and Stop, cannot be control dependent on any edge, since by design the former has no predecessors and the latter postdominates every node (being the root of the postdominator tree). There is, thus, one predicate p such that K(p) is empty. We take this predicate to be a constant true.

The role of the fictitious edge, (Start, Stop, false), is to guarantee that all nodes other than Start and Stop be control dependent on at least one edge. That is, all nodes except Start and Stop are associated with predicates p such that K(p) is not empty. Consider Head, the unique true successor node of Start. If there is no incoming edge to Head other than (Start, Head, true), then Head can only be control dependent on

+Start. Thus, Head and any other nodes equivalent to it are assigned the predicate p such that there is only one definition for it, K(p)={Start}. If K(p) is a singleton, then once p is set it remains set forever. We thus conclude that there are constant predicates identified as in the following Lemma.

Lemma 3 Constant Predicates

$$p \equiv \text{true} \quad \text{if } K(p) = \phi, \text{ or}$$

$$p \equiv \text{true} \quad \text{if } K(p) = \{\text{Start}\}$$

The decomposition of CD into R and K is purely graph-theoretic, since the process does not rely on any particular semantics of predicated execution (PE). We now turn to the meaning of R and K by describing how they are used in PE in place of graph edges (CD).

3.2 R and K as used in PE

Our aim here is to briefly sketch essential aspects of arguments showing that R and K can be used as in PE totally eliminating all branching operations. That is, the predicated and reduced code (consisting essentially of one node apart from technical Start and Stop and *unique upto topological sorting*) behaves in semantics the same as the original flow graph with conditional branching. In so doing we introduce certain notions that are useful for subsequent discussions.

Since we use K(p) to insert into nodes predicate defining operations for p, we must insure that for any such node we never have to insert more than one definition per given p. This is a direct consequence of the following lemma.

Lemma 4

A node cannot be control dependent on both true and false edge of another node.

The process of inserting defining operations using K is thus well-founded. We can also use the notation, $\pm y \in K(p)$, meaning only one edge is involved. We next want to ask, when predicates defined in this manner are used with the properties (P1) through (P3), whether the effect of control dependence has been equivalently captured. For example, once a predicate p is set true by executing the p defining operation at node y for use at node x, we must show that there cannot be another definition of p setting it false in a node present between y and x with respect to the reduced code. Our arguments must not only involve nodes in a path from y to x in the original flow graph but also those not in any path from y to x but occurring between y and x due to topological sorting.

Let p=R(x) and $+y \in K(p)$. Assume $p_y \wedge t_y \wedge \text{after}(y)$ is true. That is, the node y with the defining operation for p is enabled setting p to true. We say that a node z is *p-interfering* (with respect to +y and x), iff

1. x is control dependent on an edge of z, and

2. z occurs between y and x with respect to a topological order, $y < z < x$.

And similarly for the symmetric counterpart with $-y \in K(p)$ and $\neg t_y$. If x is not control dependent on z (Condition (1) false), then $\pm z$ cannot be a member of K(p). Hence z cannot contain p-defining operation. If x is control dependent on an edge of z then certainly there is a path from z to x and we must have $z < x$. The condition $y < z$ is simply that no ancestor of z including z is a descendant of y. We can now cite a theorem that is fundamental to the way we use R and K in PE.

Theorem PE

Let $p = R(x)$ and $+y \in K(p)$. If $p_y \wedge t_y \wedge after(y)$ then every p-interfering node z, if any, for given +y and x, is disabled. That is, $p_z \wedge at(z)$ is false where $p_z = R(z)$. Similarly for the symmetric counterpart, $-y \in K(p)$

This theorem guerantees that a predicate p=R(x) for x, once set true at one of the nodes whose edges x is control dependent on *remains* true at x because every p-interfering nodes present are disabled. This is exactly the meaning of control dependence: once +y is an execution edge, x must eventually execute. Also it is now obvious that it is a serious flaw in the semantics of defining operations when they cannot be disabled as in the old semantics described later.

3.3 Augment K

The function K as the naming isomorphism for CD described earlier is complete with respect to the usual control flow model of execution explicitly allowing branching operations. However, it is incomplete for use in PE due to (P3). Every defining operation obtained using K that simply names CD is of the form (9). Therefore, the defining operations involving literal false, like

$$p_4 = F$$

in Start of Fig. 1, would be missing. The presence of such operations is a direct consequence of Property (P3). In PE every execution involves all nodes of the original flow graph, since all of them are combined into a single node of the reduced graph. Therefore, the only way we can reproduce the semantic effect of a particular control flow path in the original graph leaving out other blocks is to guarantee that the predicates associated with excluded blocks are false before they are executed.

In other words there cannot be a predicate p such that there is a path in the original graph from Start to Stop not defining it. These, if any, must be reset initially. In order to formalize this problem. Consider the flow graph without the fictitious edge, (Start,Stop,false) that was introduced for the purpose of computing CD. Let A(x) be the set of predicates p such that there is path from x to Stop without defining operations for p. That is,

$$A(x) \equiv \{p \in P' : \exists \text{ path}(x,Stop) \text{ such that } K(p) \cap \text{path}(x,Stop) = \phi\} \qquad (14)$$

where K is the naming isomorphism of CD, P' is the set of predicates in use excluding constants of Lemma 3. The set A(Start) consists precisely of predicates that must be reset before any execution. We thus augment K by inserting -Start (denoting a reset operation) into each set K(p) for $p \in A(Start)$. With the understanding that such reset operations are to be moved subsequently to Head when appropriate.

How does one compute A(Start)? One simple but approximate scheme is based on observing that predicates satisfying certain conditions are not members of A. For example, any non-constant predicate p satisfying

$$K(p) \cap \{\pm x \in E: x \text{ pdom Head}) \neq \phi$$

is not a member of A. This condition is particularly convenient, since the postdominator relation, pdom, is already available. Unfortunately the condition is not strong enough to lead to exact A. In general A is overestimated implying presence of *useless* reset operations. As an illustration, in the example we are considering application of the above condition yields {3, 4} as an estimate of A.[1] It includes a reset operation, p_3=F, which is useless, since p_3, being defined in every path from Start to Stop, is not a member of A. Instead of seeking ways of strengthening the condition stated above, we turn to a different approach leading to an exact calculation of A at the expense of solving dataflow equations. The method used in our algorithm is arrived at by noting the problem at hand is a variant of the well known definition-use (du-)chaining problem[1] provided we take the following modified view.

- Let all operations of a block B_x be abstracted to a *point* x. A point x has *(upwards exposed) use* of the predicate assigned to B_x and can *define* several predicates as given by K(p).
- Pretend that the block Stop *uses* all (non-constant) predicates.

With this understanding the problem at hand becomes a simplified variant of du-chaining problem as formalized by the following theorem.

Theorem DU

For each block $b \in N$, except Stop, of the original flow graph (without the fictitious edge from Start to Stop) take

$$Use(b) = \{p \in P': p = R(b)\}$$
$$Def(b) = \{p \in P': \pm b \in K(p)\}$$

And for Stop

$$Use(Stop) = P'$$

1. This is clear from K(p) of table 2 and knowing pdom(1)={1,7,9}.

Algorithm RK: Given a rooted graph, (N, E, Start), compute R and K.

Postdominator relation is used in Step 1. Algorithm DU is used in Step3

1. Compute CD: Map
   Introduce a fictitious edge, [Start, Stop, false], in E.
   Let $\forall x \in N$
      pdom(x) = {y ∈ N: y postdominates x}
      ipdom(x) = the immediate postdominator of x
   for [x, y, label] ∈ E such that y ∉ pdom(x)
      Lub := ipdom(x);
      if ¬ label then x := -x; end if;
      t := y;
      while (t ≠ Lub)
         CD(t) := CD(t) ∪ {x};
         t := ipdom(t);
         end while;
      end for;
   Remove the fictitious edge, [Start, Stop, false], as required in Step 3.

2. Decompose CD into R and K: Map
   p := 1; predicates are named in sequence starting from 1
   for x ∈ N
      t := CD(x);
      if t ∈ K then
         R(x) := q such that K(q) = t;
      else
         K(p) := t;
         R(x) := p++;
         end if;
      end for;

3. Augment K:
   Perform Algorithm DU and obtain IN(Start)
   for p ∈ IN(Start)
      K(p) := K(p) ∪ {-Start};
      end for;

end of Algorithm RK.

Figure 2. Algorithm RK for computing R and K.

Algorithm DU: Given a rooted graph, (N, E, Start), R, and K at the end of Step 2 of Algorithm RK, compute IN (and OUT) of data flow equations.

$\forall b \in N$

IN(b) = Use(b) $\cup$ (OUT(b) - Def(b))

OUT(b) = $\cup$ IN(s) for s $\in$ succ(b)

where Def and Use are computed from K and R respectively.

1. Compute Def and Use

   Let P be the set of all predicates in use except constants:

   P' = {p $\in$ domain (K): K(p) $\neq \phi$}

   For each block b Use(b) is the upwards exposed use of a predicate, which in our case is simply the associated predicate R(b).

   Pretend Use(Stop) = P'. Let N' = N - {Stop}.

   ```
   for b ∈ N'
      Def(b) := {p ∈ P': b ∈ K(p) ∨ -b ∈ K(p)};
      Use(b) := {p ∈ P': p = R(b)};
      end for;
   Use(Stop) := P';
   ```

2. Solve data flow equations stated above for IN and OUT

   A simple iterative method is shown below . However, calculations are better performed in depth first order.

   ```
   IN := Use;     initial estimate
   change := true;
   while change
      change := false;
      for b ∈ N'
         OUT(b) := ϕ;
         for s ∈ succ(b)
            OUT(b) := OUT(b) ∪ IN(s);
            end for;
         old := IN(b);
         IN(b) := Use(b) ∪ (OUT(b) - Def(b));
         if old ≠ IN(b) then change := true; end if;
         end for;
      end while;
   ```

end of Algorithm DU.

Figure 3. Algorithm DU for computing IN(Start)

where R and K are the decomposition of CD as in Lemma 2 and P' is the set of predicates leaving out constants of Lemma 3. Let IN be the solution of the (data flow) equations,

$$IN(b) = Use(b) \cup (OUT(b) - Def(b))$$
$$OUT(b) = \cup\, IN(s) \quad \forall s \in succ(b)$$

Then the augment set A(Start) we seek is

$$A(Start) = IN(Start)$$

We shall not bother proving this theorem. Finally for the sake of completeness we turn to the problem of computing CD.

3.4 Computing CD

The method used is from the PDG work[3] and is based on the following theorem on control dependence and postdominator relation.

Theorem CD

Let (x, y, label) be an edge such that y does not postdominate x. The nodes control dependent on this edge are those and only those of the unique path starting (excluding the first) from the immediate postdominator of x, ipdom(x), to y in the postdominator tree.

Viewing the postdominator tree as a join-semilattice with Stop as *Top* we know that a unique least upperbound, lub(x,y), exists for any pair x and y. In particular, since y does not postdominate x, it is not "above" x in the lattice. Hence, there is a unique nonempty path from lub(x,y) to y in the postdominator tree. The theorem stated above includes a further simplification due to a lemma in [3] showing that lub(x,y) in this case is simply the immediate postdominator of x, ipdom(x). This brief argument suggests that the theorem is not vacuous. An argument for its correctness is found in [3].

3.5 Algorithm for R and K

Our algorithm for computing R and K is shown in Fig. 2. In expressing algorithms we employ a language allowing direct manipulations of abstractions like set, tuple, map, etc., as in SETL[11]. Additional advantage in specifying algorithms in this form is that they are easily (literally) translated to SETL and executed. As discussed in text Algorithm RK consists of three steps. In Step 1, given a rooted control flow graph, (N, E, Start), we compute the control dependence function CD of (8).Then in Step 2 we decompose CD into the naming isomorphism K (unique upto renaming of predicates) and the assignment function R of (9). Finally in Step 3 we augment K to fully satisfy Property (P3).

Briefly, Step 1 is a literal translation of Theorem CD. Step 2 constructs R and K from CD such that for all nodes Lemma 2 is satisfied. Clearly this is the requirement for this step. In the example we are considering, at the end of Step2 we obtain the solutions, R and K, shown in Table 2. Note that K at this point is incomplete and must be augmented as in Step 3. Observe also that K prescribes certain predicates to be constant, true. These are identified as in Lemma 3.

From K(p) of Table 2 we see that both p_1 and p_7 are constants. From R(x) we know that p_7 is associated with Start and Stop, and p_1 with blocks 1 and 7. We need not bother further with such constant predicates.

Table 2. Solutions R and K at the end of Step 2.

block x	1	2	3	4	5	6	7	8	9
predicate R(x)	1	2	5	4	3	6	1	7	7

predicate p	1	2	3	4	5	6	7
K(p)	{8}	{1}	{-2, 3}	{2}	{-1}	{1, 3}	ϕ

As discussed earlier K obtained in Step 2 is incomplete for use in PE and needs to be augmented as in the last step.

Finally Step 3 relies on Algorithm DU for computing the augment set A(Start) exactly. Algorithm DU, as shown in Fig. 3, is a straight forward application of Theorem DU yielding the result IN(Start) that we seek. The latter is used in Step 3 of Algorithm RK to augment K. The result produced is optimal, in particular, with respect to the number of defining operations, since the set A is calculated exactly insuring that there are no useless defining operations.

Parenthetically Algorithm DU shows a simple iterative scheme like that of Aho et al.[1] for solving the equations of Theorem DU. As is well known the number of iterations performed, and hence, the time complexity of this step, is highly dependent on the order of blocks examined (in "for b $\in$ N' " loop of Step 2 of Algorithm DU). Since information is being propagated upwards starting from Stop through all blocks to the end goal, Start, the preferable order is depth-first. The subject of how to solve such equations efficiently is outside the scope of this paper, however. We have not bothered to include such complications in Algorithm DU.

For the example we have been considering Algorithm DU yields

$$IN(Start)=\{4\}$$

As a result, at the end of Step 3 we obtain the final result, the augmented K, shown in Table 3.

Table 3. Augmented K.

predicate p	1	2	3	4	5	6	7
K(p)	{8}	{1}	{-2, 3}	{2,-8}	{-1}	{1, 3}	ϕ

Note the insertion of reset operation in K(4) when compared with the intermediate result for K in Table 2.

3.6 Features of Algorithm RK

Here we address several features of our algorithm and a way of visualizing the result.:

- Results produced are optimal with respect to number of predicates in use and that of defining operations.
- The algorithm is general in that any flow graph including cyclic and irreducible can be tackled.
- The time complexity of the entire algorithm is dominated by that of computing control dependence. (Step 1 of Algorithm RK performed as in PDG.) It is $0(N^2)$.

3.6.1 Number of Predicates and Defining Operations

Since predicates are simply names of the equivalence classes under control dependence (CD) the number of predicates in use is precisely that required. There are no "redundant" predicates as long as every edge of the graph is potentially an execution edge. The same is true with respect to defining operations. Since the augment set A(Start) is computed exactly for a given graph, there are no useless defining operations. Thus, for a given graph, our algorithm is *optimal* with respect to both the number of predicates and of defining operations in use.

As usual in such problems the word, optimal, here has to be understood in a limited sense. As an illustration consider K(3) of Table 3. Suppose under a global analysis, say, in the manner of Alpern et al.[2], of the code not shown in Fig. 1 we can prove $t_2=t_1$ statically. Then we conclude K(3)=K(6) and therefore one predicate may be eliminated. In fact, in this case the block B_4 is unreachable and we should have processed a different flow graph that takes into account such optimizations.

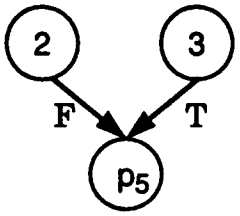
Note that presence of redundant predicates would unnecessarily increase the number of predicate registers required. The presence of useless defining operations, on the

other hand, would unnecessarily increase output dependence arcs that constrain scheduling.

3.6.2 Pictorial Representation

The results, R and K, can be visualized by the following pictorial representation. Consider R of Table 2. For each entry, say, R(5)=3, we associate predicate 3 with block 5 as

Similarly, for each entry of Table 3 for the augmented K, say, K(3)={-2,3}, we draw the labelled arcs as



Completing these steps we obtain the full graph shown in Fig. 4a. This result is to be compared with the corresponding PDG result (Fig. 4b) from [3]. There are essential differences, namely (1) absence of predicate-to-predicate arcs and (2) presence of false arcs from Start in our result. Region-to-region arcs in PDG come about due to "factoring" (not required in our method) discussed there. Presence of additional arcs from Start represent reset operations (not required in PDG) in our method. It is obvious that our predicates are not simply regions nodes of PDG.

3.6.3 Generality

A close scrutiny of Algorithm RK shows that hardly any restriction is required on the character of the flow graph. And so does the calculation of postdominator relation and the method of solving du-chaining problem. Hence, it is not surprising that Algorithm RK can tackle arbitrary flow graph including cyclic as well as irreducible. We will not pursue this further here, however, since our primary need for predicated execution is to overlap iterations (that is, software pipeline) of an innermost natural loop with if statements. By definition such a loop is associated with a single backedge, a single entry, and an acyclic body.

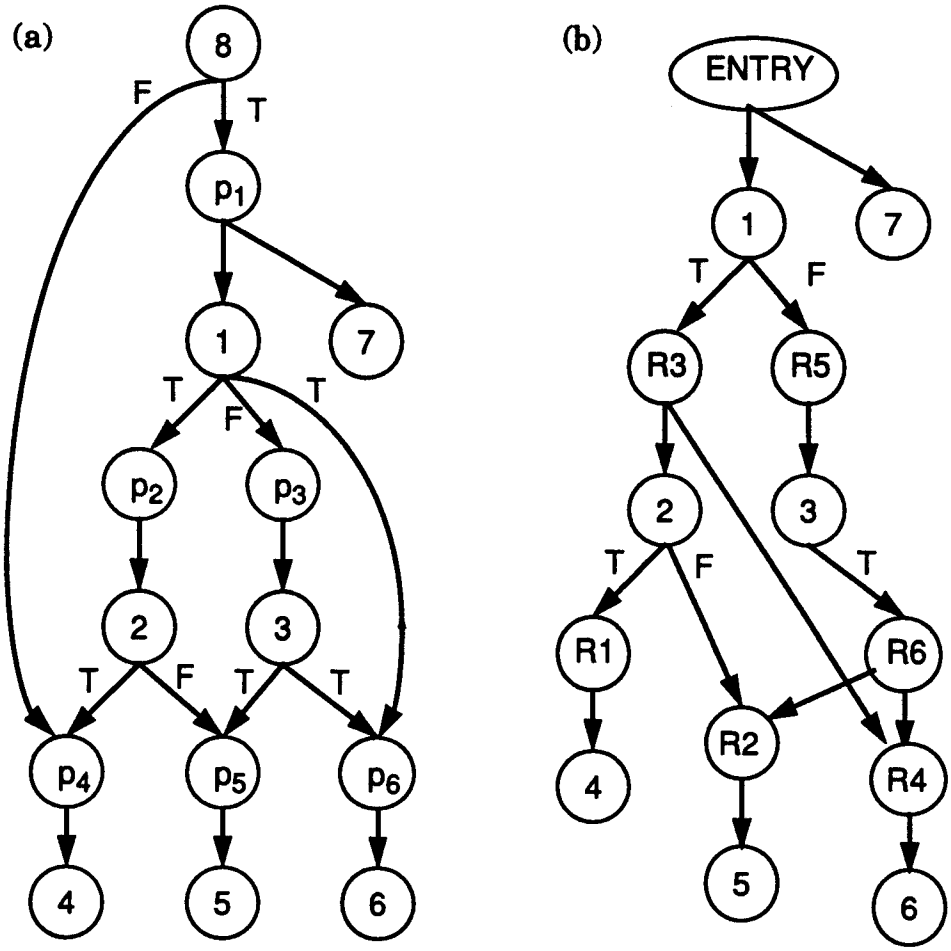


Figure 4. Compare RK (left) with PDG (right).

3.6.4 Time Complexity

At the level of abstraction shown in Fig. 1 the time complexity of Algorithm RK is easily analyzed. Given that the operations like set membership ($\in$ and $\notin$), or union ($\cup$) are performed in a way independent of problem size, viz., the number of nodes (#N) and that of edges (#E), we only have to examine loops. We will use the notation like #N to mean the cardinality of a set N.

As in PDG [3] introduce a set S

$$S \equiv \{[x,y,label] \in E: y \text{ does not postdominate } x\}$$

Edges in S are precisely those examined in "for" loop of Algorithm RK Step 1 so that the number of iterations of this loop is #S. But $\#S < \#E < 2\ \#N$. The number of iterations of the nested "while" loop, on the other hand, cannot exceed the maximum height of the postdominator tree, which in turn cannot be greater than #N. Hence the

time complexity of Step 1 is $0(\#N^2)$. Note that the postdominators required in this step can be obtained in almost linear time, $0(N\alpha(N))$, using the algorithm of Lengauer and Tarjan[7] for dominators with all graph edges reversed.

That of Step 2 is clearly determined by the number of iterations of "for" loop, which is #N. Setting aside the time taken to compute IN(Start) the complexity of Step 3 is given by the number of iterations of "for" loop, which cannot be greater than the total number of predicates in use, #P. We already know that #P is precisely the number of equivalence classes under which N has been partitioned. Hence we have #P < #N.

Examining Algorithm DU in order to set a time bound. Step 1 can be performed in 0(#N) time. Step 2 is somewhat involved. The innermost "for s" loop is not a loop in the sense that at most two iterations are involved and this number is clearly problem size independent. If we take the specification literally as a code, then the outermost "while" loop is bounded above by #N, since every IN(b) can only grow (does not decrease) and none of it can grow larger than P'. Thus Step 2 as given has time complexity of $0(\#N^2)$.

However, as mentioned earlier, one can perform "for b" loop in depth first order a la Tarjan[12] in time linear in #N and effectively avoid the outer "while" loop. In any case we conclude that the time complexity of entire Algorithm RK is $0(\#N^2)$.

4 Comparison with Earlier Results

To our knowledge predicated execution in the form we are discussing has been used for the first time at Cydrome as in the architecture of Cydra 5[10] and a Cydrome compiler[4]. Although the current work was motivated primarily to understand the details of this novel approach, it led to a new scheme that cures certain deficiencies in the old approach subsequently discovered, namely:

- Incorrect results for certain graphs.
- Presence of useless reset operations.

This is achieved by having to (1) redefine semantics of predicate operations and (2) design a new algorithm. It also turned out that our algorithm is simpler conceptually and more efficient in time complexity. However, since the old algorithm has not been published in detail, we will not dwell on algorithmic details of the old approach.

4.1 The Old Semantics of Predicate Operations

The semantics of the predicate defining operations (stuff and stuffbar) in Cydra 5 architecture are different from ours of (3) and (4). For stuff operation (1) the meaning is

$$\begin{aligned} p_y.new &= t_x.old \wedge p_x.old \\ t_x.new &= t_x.old \\ p_x.new &= p_x.old \end{aligned} \qquad (16)$$

Similarly for stuffbar operation (2) the meaning is

$$\begin{aligned} p_y.new &= \neg t_x.old \wedge p_x.old \\ t_x.new &= t_x.old \\ p_x.new &= p_x.old \end{aligned} \qquad (17)$$

Notice that in this scheme the target predicate p_y is modified regardless of whether the source predicate p_x is true or not. This is counter-intuitive in that two stuff operations with a common target do not commute even if they are associated with complementary source predicates. Our approach is more natural in this respect, since one or the other behaves as No Operation and therefore the (semantically) correct operation of one operation does not depend on where the other operation is. In other words in the old scheme it is impossible to disable (or nullify) a stuff operation. There are penalties associated with this inability as elaborated below.

Consider the example of the reduced code (7) with our solution shown in Fig. 1. When examined under the old semantics, (16) and (17), we obtain the behavior for the case t_1=false and t_2=true as sketched in Table 4. Clearly the code is broken in semantics, since p_3 is false at B_5. The defining instruction $p_3=t_2$ of block B_2 (not enabled in this behavior) interferes with $p_3=t_3$ of the enabled block B_3, since the former occurs after the latter. Since commuting B_2 and B_3 is another possible linearization, is it possible to correctly solve the placement problem subject to a particular choice of topological sort? If so, the penalty implied by the old semantics is merely giving up invariance under topological sorting? The answer appears to be negative as explained subsequently.

4.2 The Old Algorithm

Setting aside algorithmic complexities one must in the old approach address the problem of compensating for the inability to disable defining operations, stuff and stuffbar. As observed earlier, as a defining operation is moved up farther away from a block using it, it is more likely that there will be interfering defining operations forcing predicates false in spite of being "disabled." If we compare the two semantics, old and new, their difference occurs when the source predicate p_x is false forcing the target predicate to be false, p_y.new=false.

Table 4. A Behavior of Reduced Code under Old Semantics

p_x	after(B_x)	p_2	p_3	p_4	p_5	p_6	Remark
T	Start	X	X	F	X	X	
T	B_1	F	X	F	T	F	$t_1 = F$
p_5	B_3	F	T	F	T	T	$t_3 = T$
p_2	B_2	F	F	F	T	T	
p_4	B_4						
p_3	B_5						$p_3 = F$ at(B_5)!
p_6	B_6						
T	B_7						

This has a different sematic effect only when we have p_y.old=true, since otherwise p_y stays unchanged (as if disabled). One approach is, thus, reset all predicates false in the beginning and set p_y true in every immediate predecessor of node y.

The approach found is consistent with this observation and is roughly based on the following strategies: Consider a graph corresponding to the body of an innermost natural loop, which by definition is acyclic.

1. Establish an order by topological sorting.
2. Examining blocks in the topological order assign predicates putting blocks x and y such that x dominates y and y postdominates x under a common predicate.
3. For each conditional flow edge (x,y,label) put defining operation p_y=stuff(t_x)&p_x in block x, if label is true. Similarly, stuffbar if label is false.
4. For unconditional edge (x,y) put defining operation p_y=stuff(true)&p_x in block x.
5. At Head node reset all predicates in use except those defined at Head.

The actual algorithm used is considerably more complex than suggested above. It is clear that the number of predicates in use must agree in both methods, old and new, since this is independent of semantics of defining operations, being purely a graph-theoretic property. The differences arise in the number of defining operations and their placement, since this depends, in addition, on the semantics of stuff.

For instance, when the example we have been using is processed under the old algorithm we get the result shown in Fig. 5. (Start and Stop nodes required in the new algorithm are not used in the old.) In the result shown all reset operations present in Head node are useless except p_4=F. In addition p_6=T in block B_2 is useless. Comparing this result with our earlier result shown in Fig. 1 we observe differences summarized in Table 5 where predicates are counted leaving out constant predicates of Lemma 3, defining operations (stuff and stuffbar) include those of reset operations,

and output dependence arcs counted are only those between predicate defining operations.

Table 5. Comparing Predicated Code in Two Methods.

Features	New	Old
Predicates	5	5
Stuff Operations	8	11
Output Dep. Arcs	3	6

The number of dependence arcs is an important measure of parallelism, fewer arcs leading to more parallel code. Our algorithm produces inherently more prallel code by minimizing the number of defining operations thereby reducing arcs.

Parenthetically, counting of the output dependence arcs in the table is conservative, since the effect of predicates associated with defining operations is not taken into account. Referring to the new result in Fig. 1 the dependence arc between p_3 in block B_2 and p_3 in block B_3 , for example, is not needed, since the predicates, p_2 and p_5, associated with respective defining operations satisfy, $p_2 \wedge p_5$ = false, meaning one or the other is disabled. Thus, under the new semantics of PE dependence analysis taking predicates into account one can further reduce arcs by eliminating such spurious ones. In other words, the old semantics, in which defining operations cannot be disabled, precludes the benefit of such analysis.

If the elimination of useless defining operations is the only gain, then the effect of the new algorithm is merely an improvement in performance of scheduled code. This is actually not the case. As discussed earlier the old algorithm, unlike the new, produces solutions that are dependent on a particular topological sort. What would happen if a "symmetric" flow graph with no intrinsic order is tackled?

Consider the flow graph of Fig. 6, which is our earlier example, Fig. 1, with the edge, (3,7,false), removed and a new edge, (3,4,false), added. (Also blocks 6 and 7 are combined due to "branch optimization".) The solution produced by the old algorithm (with respect to a particular order, $B_1B_3B_2B_4B_5B_6$) is shown in the same figure. Apart from the useless reset operations (p_3=F and p_4=F) present at Head this solution agreew with that of the new algorithm.

It is easy to see that under the old semantics having p_2=false (corresponding to the behavior for t_1=false) at B_2 results in both predicates p_4 and p_5 false so that neitherB_4 nor B_5 can be enabled.

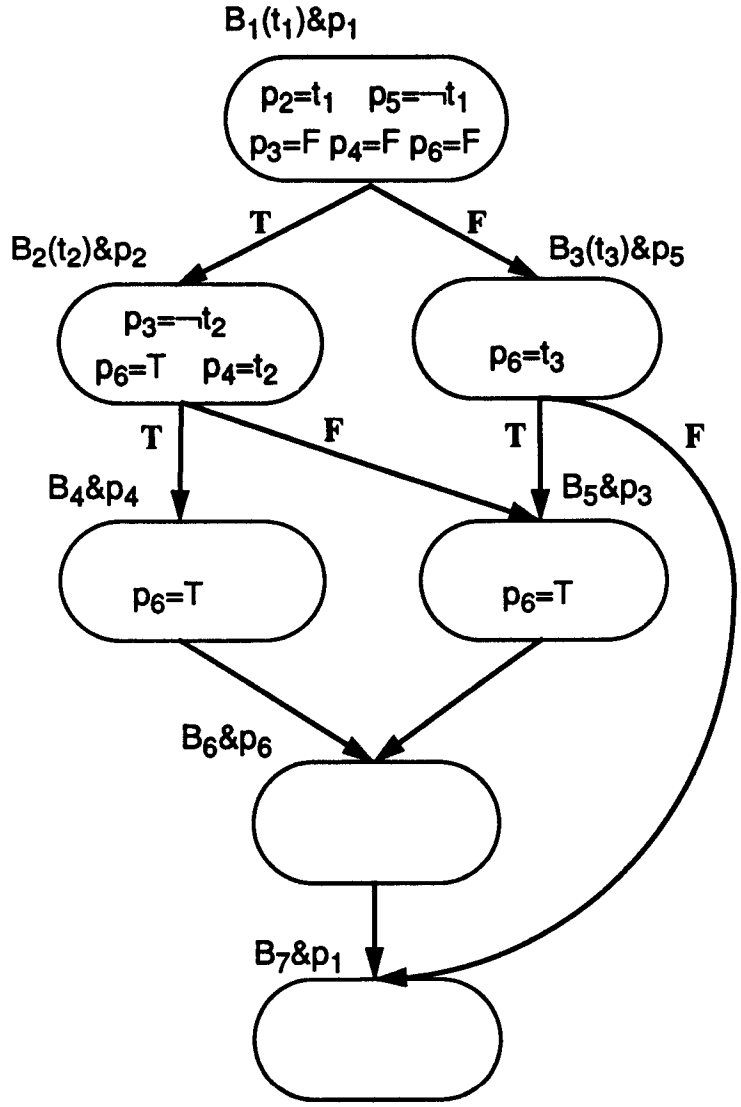


Figure 5. The flow graph of Fig. 1 predicated using the old method.

This erroneous behavior cannot be cured by commuting B_2 and B_3 in the reduced code. If B_2 is ordered to occur after B_3 in the reduced code, then in the behavior for t_1=true having p_5 false at B_3 leads to the same erroneous condition of having both p_4 and p_5 false before their use. In fact, once the semantics of predicate defining operations are taken to be those of the old approach (15) and (16), a correct algorithm

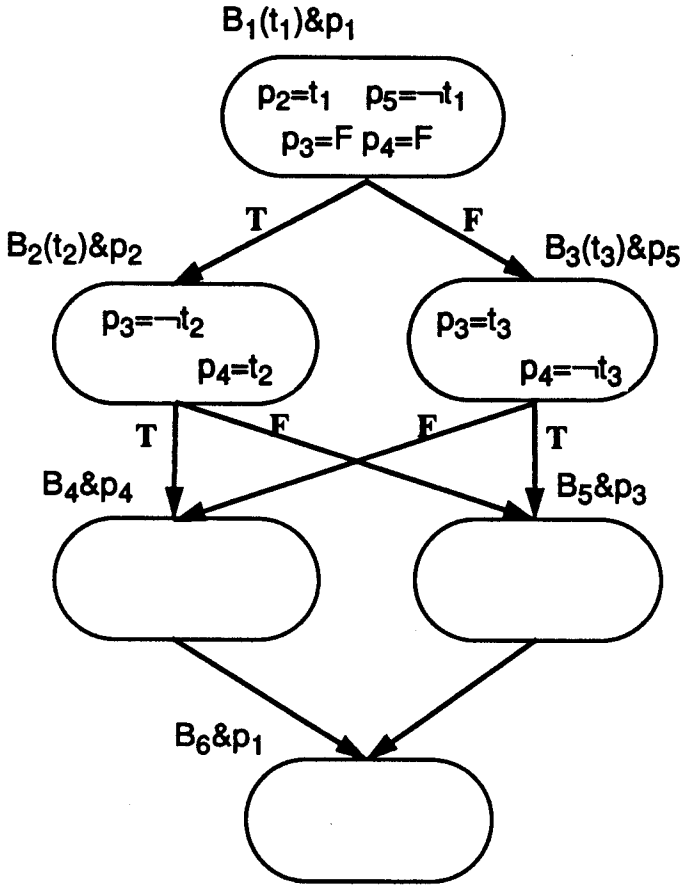


Figure 6. Another flow graph predicated using the old method.

for PE does not appear feasible, not to speak of discovering the precise conditions on graph characteristics under which the old algorithm produces incorrect results. The new algorithm cures this deficiency by choosing the semantics that guarantees code, including predicate defining operations, of a disabled block to behave as No Operation.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Alpern, B., Wegman, M. N., and Zadeck, F. K. Detecting Equality of Variables in Programs. In *Proc. of the Fifteenth Annual ACM SIGACT-SIGLPLAN Symp. on Principles of Prog. Lang.* (January 1988) 1-11.

[3] Ferrante, J., Ottenstein, K., and Warren, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9,3 (July 1987) 319-349.

[4] Dehnert, J. C., Hsue, P. Y. T., and Bratt, J. P. Overlapped Loop Support in the Cydra 5. In *Proc. ASPLOS III,* April 1989.

[5] Lam, M., Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. of the SIGPLAN '88 Conf. on Prog. Lang. Design and Impl.* (June 1988) *318-328.*

[6] Lamport, L. Control Predicates Are Better Than Dummy Variables For Reasoning About Programs. *DEC SRC Report* 11 (May 1986),

[7] Lengauer, T., and Tarjan, R. E. A Fast Algorithm for Finding Dominaotrs in Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (July 1979) 121-141.

[8] Park, J. C.H. Formal Aspects of Predicated Execution. *HPL Technical Report* (In preparation.)

[9] Rau, B. R., and Glaeser, C. D. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proc. of the 14th Annual Microprogramming Workshop,* October 1982.

[10] Rau, B. R., Yen, D. W. L., Yen, W., and Towle, R. A. The Cydra 5 Departmental Supercomputer, Design Philosophies, Decisions, and Trade-offs. IEEE Computer (January 1989) 12-35.

[11] Schwartz, J. T., et al. *Programming with sets: An Introduction to SETL.* Springer-Verlag, 1986.

[12] Tarjan, R. E. Depth First Search and Linear Graph Algorithms, *SIAM J. Computing* 1, 2 (1972) 146-160.